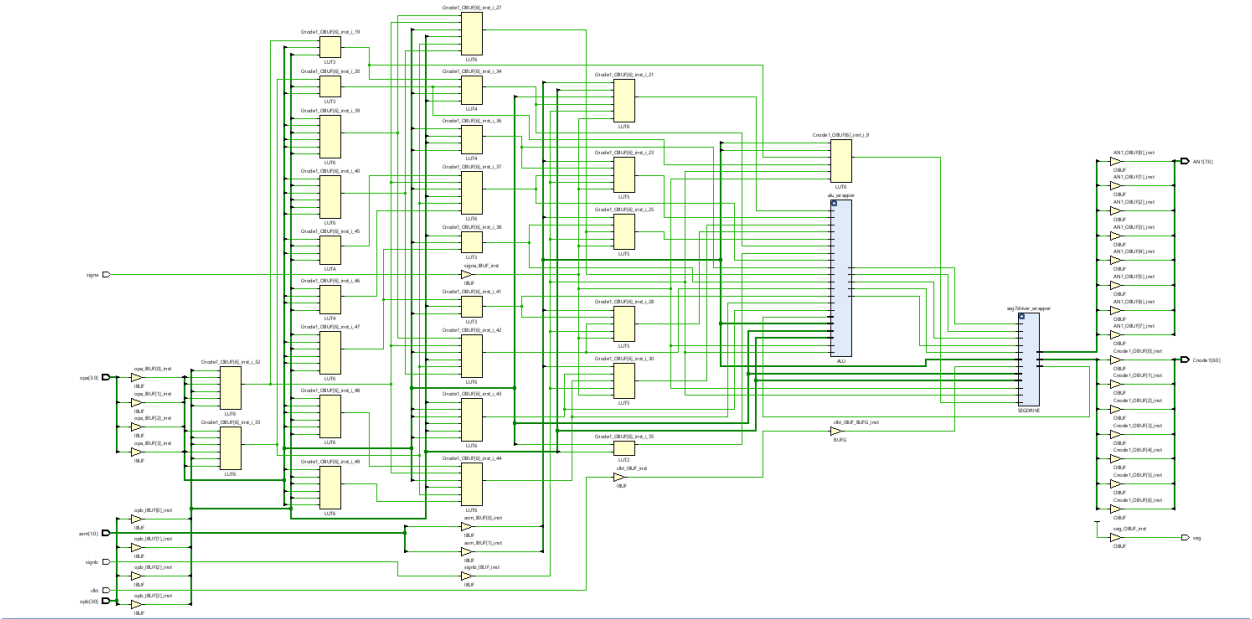
ECE3300L Lab7 Group H Report (Sherwin Sathish & Mohamed Hamida)

SCHEMATIC():



Xdc for top.v:

```
## This file is a general .xdc for the Nexys A7-100T
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clkt}];# IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clkt}];


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { opb[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { opb[1] }]; #IO_L3N_T0_DQS_EMCCLK_14
Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { opb[2] }]; #IO_L6N_T0_D08_VREF_14
Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { opb[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { signb }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { opa[0] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { opa[1] }]; #IO_L17N_T2_A13_D29_14
Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { opa[2] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { opa[3] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { signa  }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { asm[0] }]; #IO_L15P_T2_DQS_RDWR_B_14
Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { asm[1] }]; #IO_L23P_T3_A03_D19_14
Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14
Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14
Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## LEDs
```

```
#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { sumbit[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { sumbit[1] }]; #IO_L24P_T3_RS1_15
Sch=led[1]
#set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { sumbit[2] }]; #IO_L17N_T2_A25_15
Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { sumbit[3] }]; #IO_L8P_T1_D11_14
Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { coutbit }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { opc[0] }]; #IO_L18N_T2_A11_D27_14
Sch=led[5]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { opc[1] }]; #IO_L17P_T2_A14_D30_14
Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { opc[2] }]; #IO_L18P_T2_A12_D28_14
Sch=led[7]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { opc[3] }]; #IO_L16N_T2_A15_D31_14
Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { opc[4] }]; #IO_L14N_T2_SRCC_14
Sch=led[9]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { opc[5] }]; #IO_L22P_T3_A05_D21_14
Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { opc[6] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14
Sch=led[11]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { opc[7] }]; #IO_L16P_T2_CSI_B_14
Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { signc1 }]; #IO_L22N_T3_A04_D20_14
Sch=led[13]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14
Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14
Sch=led[15]

## RGB LEDs
#set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { LED16_B }]; #IO_L5P_T0_D06_14
Sch=led16_b
#set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { LED16_G }]; #IO_L10P_T1_D14_14
Sch=led16_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14
Sch=led16_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15
Sch=led17_b
#set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
#set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14
Sch=led17_r

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[6] }]; #IO_L24N_T3_A00_D16_14
Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[5] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[4] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[2] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[1] }]; #IO_L19P_T3_A10_D26_14
Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[0] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { seg }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN1[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN1[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN1[2] }]; #IO_L24P_T3_A01_D17_14
Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN1[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN1[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN1[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN1[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN1[7] }]; #IO_L23N_T3_A02_D18_14
Sch=an[7]
```

Code Detail:

`timescale 1ns / 1ps

```
module ALU(
    input [3:0] opa,
    input [3:0] opb,
    input signa,
    input signb,
    input [1:0] asm,
    output [7:0] opc,
    output signc1
    );
    reg [7:0] ropc=0;
    reg signc=0;
    always@(*)
    begin
    case(asm)
            2'b00:
            begin
            ropc <= 0;
            signc <=0;
            end

            2'b01://ADD
            begin
            if(opa==0 && opb==0)
              begin
              ropc=0;
              signc=0;
              end
              else if(signa==0 && signb==1)
                begin
                    if(opa<opb)
                    begin
                    ropc <= opb-opa;
                    signc <=1;
                    end
                   else if(opa>opb)
                    begin
                    ropc <= opa-opb;
                    signc <=0;
                    end
                    else
                    begin
                    ropc <=0;
                    signc <=0;
                    end
                end
               else if(signa==1 && signb==0)
                begin
                    if(opa<opb)
                    begin
                    ropc <= opb-opa;
                    signc <=0;
                    end
                   else if(opa>opb)
                    begin
```

```
        ropc <= opa-opb;
        signc <=1;
        end
        else
        begin
        ropc <=0;
        signc <=0;
        end
    end
    else if(signa==1 && signb==1)
    begin
    ropc <=opa+opb;
    signc <=1;
    end
    else
    begin
    ropc <=opa+opb;
    signc <=0;
    end
end

2'b10://SUBTRACT
begin
if(opa==0 && opb==0)
  begin
  ropc=0;
  signc=0;
  end
   else if(signa==0 && signb==0)
    begin
        if(opa<opb)
        begin
        ropc <= opb-opa;
        signc <=1;
        end
       else if(opa>opb)
        begin
        ropc <= opa-opb;
        signc <=0;
        end
        else
        begin
        ropc <=0;
        signc <=0;
        end
    end
   else if(signa==1 && signb==1)
    begin
        if(opa<opb)
        begin
        ropc <= opb-opa;
        signc <=0;
        end
       else if(opa>opb)
```

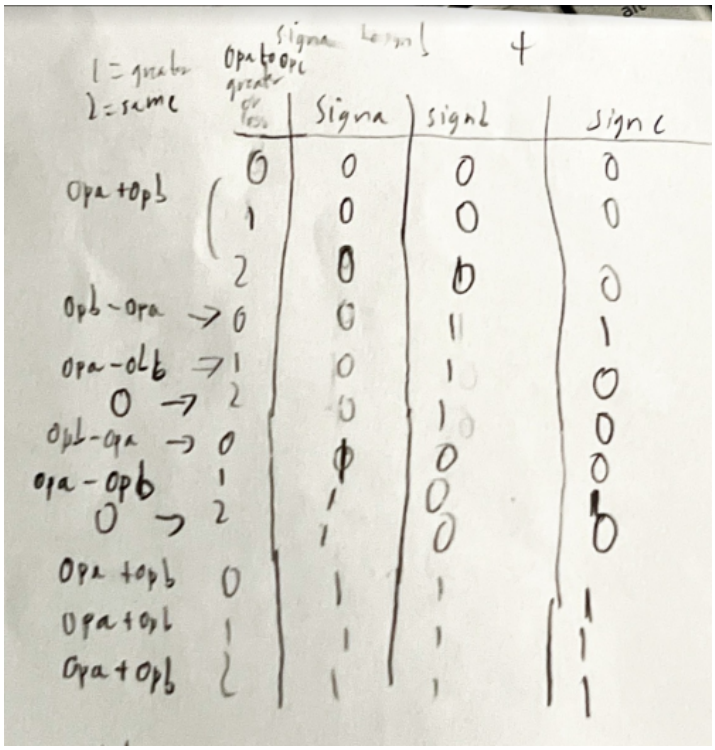
```
                begin
                ropc <= opa-opb;
                signc <=1;
                end
                else
                begin
                ropc <=0;
                signc <=0;
                end
              end
              else if(signa==1 && signb==0)
              begin
              ropc <=opa+opb;
              signc <=1;
              end
              else
              begin
              ropc <=opa+opb;
              signc <=0;
              end
            end

            2'b11: //MULTIPICATION
            begin
             ropc = opa * opb;
              if(opa==0 && opb==0)
              begin
              ropc=0;
              signc=0;
              end
               else if(signa==1 && signb==1)
                   signc <= 0;
              else if(signa==0 && signb==0)
                   signc <= 0;
              else
                   signc <= 1;

            end

        endcase
    end
    assign opc = ropc;
    assign signc1 = signc;
endmodule



`timescale 1ns / 1ps
module alu_tb();
    reg [3:0] opatb;
    reg [3:0] opbtb;
    reg signatb;
    reg signbtb;
    reg [1:0] asmtb;
```

```
    wire [7:0] opctb;
    wire signctb;

    ALU alu_tb(
        .opa(opatb),
        .opb(opbtb),
        .signa(signatb),
        .signb(signbtb),
        .asm(asmtb),
        //output to 7seg
        .opc(opctb),
        .signc1(signctb)
    );
    initial
        begin: TEST
            asmtb = 3;
            opatb = 4;
            opbtb = 2;
            signatb = 0;
            signbtb = 0;

            #100

            asmtb = 2;
            opatb = 4;
            opbtb = 4;
            signatb = 0;
            signbtb = 1;

            #100

            asmtb = 1;
            opatb = 9;
            opbtb = 1;
            signatb = 0;
            signbtb = 0;

            #100
            asmtb = 3;
            opatb = 1;
            opbtb = 9;
            signatb = 1;
            signbtb = 0;
            #1000
            $finish;
        end
endmodule
```

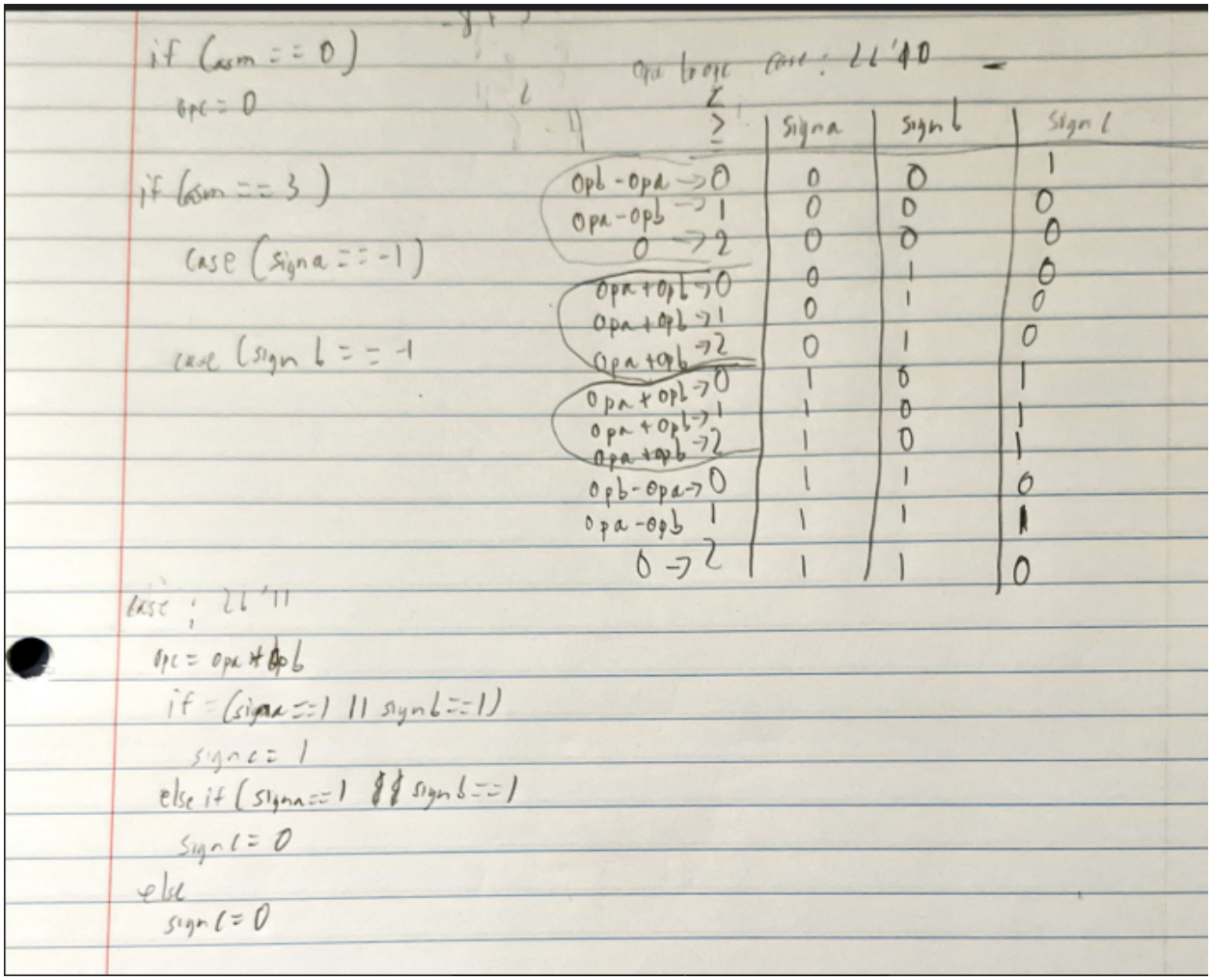
The alu code takes 2 single digit numbers and either adds, subtracts, or multiplies them to yield a 2 digit answer. To solve this we created truth tables to simplify the algorithm. We compared the signs first, then we set opa as the main digit, and created many cases based on opa compared to opb. At the end of the code we yield the opc and the signc values based on

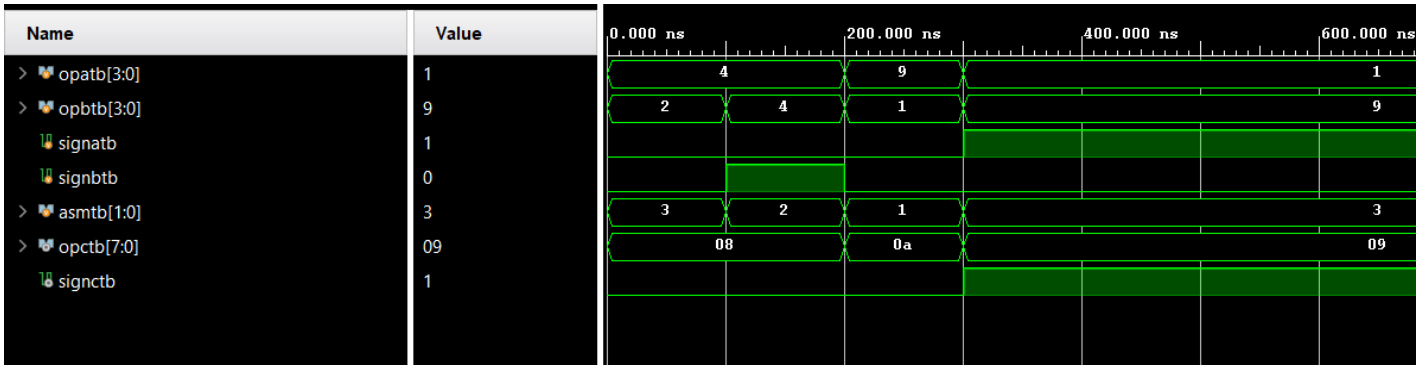
the different cases. The truth tables were used for addition and subtraction, but multiplication was simple because a size comparison between opa and opb wasn't necessary. The truth tables and testbench are shown below:

ADDITION:



SUBTRACT AND MULTIPLY:


if (asm == 0)
opc = 0
if (asm == 3)
case (signa == -1)
case (signb == -1
case : 2'b11
opc = opa * opb
if (signa == 1 || signb == 1)
signc = 1
else if (signa == 1 && signb == 1)
signc = 0
else
signc = 0
opa logic case : 2'b10
Signa
Signb
Signc
opb - opa → 0
opa - opb → 1
0 → 2
opa + opb → 0
opa + opb → 1
opa + opb → 2
opa + opb → 0
opa + opb → 1
opa + opb → 2
opb - opa → 0
opa - opb → 1
0 → 2


Name
Value
opatb[3:0]
1
opbtb[3:0]
9
signatb
1
signbtb
0
asmtb[1:0]
3
opctb[7:0]
09
signctb
1
0.000 ns
200.000 ns
400.000 ns
600.000 ns
4
9
1
2
4
1
9
3
2
1
3
08
0a
09

```
module bin2bcd(
  input [7:0] bin,

  output reg [7:0] bcd
  );

integer i;

always @(bin) begin
   bcd=0;
   for (i=0;i<6;i=i+1) begin



  // else
  // begin
        //Iterate once for each bit in input number
   if (bcd[3:0] >= 5) bcd[3:0] = bcd[3:0] + 3;            //If any BCD digit is >= 5, add three
        if (bcd[7:4] >= 5) bcd[7:4] = bcd[7:4] + 3;

        bcd = {bcd[7:0],bin[5-i]};
   //end

        end                        //Shift one bit, and shift in proper bit from input
   end
endmodule

`timescale 1ns / 1ps
module bin2bcd_tb(
   );
   reg [31:0]bin_tb;
   wire [31:0] bcd_tb;

   bin2bcd COMP (
   .bin(bin_tb),
   .bcd(bcd_tb)
   );
   initial
      begin
      bin_tb = 12;
      //$finish
      end
```

endmodule

The purpose of this code is to implement the double dabble algorithm. The “double dabble” algorithm is commonly used to convert a binary number to BCD. The binary number is left-shifted once for each of its bits, with bits shifted out of the MSB of the binary number and into the LSB of the accumulating BCD number. After every shift, all BCD digits are examined, and 3 is added to any BCD digit that is currently 5 or greater. This works because every left shift multiplies all BCD digits by two. Since BCD digits cannot exceed nine, a pre-shift number of five or more would result in a post-shift number of ten or more, which cannot be represented. Adding three to any BCD digit greater than five does two things: first, at the next shift, the 3 that was added becomes 6, and that accounts for the difference in binary and BCD codes. This code takes the opc output of the alu and converts it into BCD.

```
`timescale 1ns / 1ps

module segdisplaydriver(
    input [31:0] inDigit,
    output reg [6:0] Cnode,
    output dp,
    output reg [7:0] AN,
    input nexysCLK, // 100MHz
    output reg divided_clk = 0 // 10kHz => 10ms period, 0.5ms ON, 0.5ms OFF
    );
    reg [3:0] singledigit = 0;
    reg [3:0] refreshcounter = 0;
    // Calculate division value = 100MHz / (2 * desired frequency) - 1 => 10kHz => 4999

    localparam div_value = 25000;

    integer counter_value = 0;

    always @(posedge nexysCLK)
    begin
        if (counter_value == div_value)  // For every (div_value) clock cycles, reset counter back to 0
            counter_value <= 0; // Use <= for parallel & same time, = for sequential - one after the other
        else
            counter_value <= counter_value + 1;
    end

    // divide clock
    always @(posedge nexysCLK)
    begin
        if (counter_value == div_value)
            divided_clk <= ~divided_clk; // Flip signal
        else
            divided_clk <= divided_clk; // Keep signal the same
    end
    /*CLOCK DIVIDER CODE*/

    always @(posedge divided_clk)
```

```
begin
   refreshcounter <= refreshcounter + 1;
end

always @(refreshcounter)
   begin
      case(refreshcounter)
         4'b0000: singledigit = inDigit[3:0];    // digit 1 value (right digit)
         4'b0001: singledigit = inDigit[7:4];    // digit 2 value
         4'b0010: singledigit = inDigit[11:8];    // digit 3 value
         4'b0011: singledigit = inDigit[15:12];    // digit 4 value
         4'b0101: singledigit = inDigit[19:16];    // digit 5 value missing 4'b0010
         4'b0110: singledigit = inDigit[23:20];    // digit 6 value
         4'b0111: singledigit = inDigit[27:24];    // digit 7 value
         4'b1000: singledigit = inDigit[31:28];    // digit 8 value (left digit)
      endcase
   end

always @(refreshcounter)
begin
   case(refreshcounter)
      4'b0000: AN = 8'b11111110;     // digit 1 ON (right digit)
      4'b0001: AN = 8'b11111101;     // digit 2 ON
      4'b0010: AN = 8'b11111011;     // digit 3 ON
      4'b0011: AN = 8'b11110111;     // digit 4 ON
      4'b0100: AN = 8'b11101111;     // digit 5 ON
      4'b0101: AN = 8'b11011111;     // digit 6 ON
      4'b0110: AN = 8'b10111111;     // digit 7 ON
      4'b0111: AN = 8'b01111111;     // digit 8 ON (left digit)
      default: AN = 8'bZZZZZZZZ;
   endcase
end

assign dp = 1'b1;
always@(singledigit)
   begin
      case (singledigit)
         4'd0: Cnode<= 7'b0000001; //0
         4'd1: Cnode<= 7'b1001111; //1
         4'd2: Cnode<= 7'b0010010; //2
         4'd3: Cnode<= 7'b0000110; //3
         4'd4: Cnode<= 7'b1001100; //4
         4'd5: Cnode<= 7'b0100100; //5
         4'd6: Cnode<= 7'b0100000; //6
         4'd7: Cnode<= 7'b0001111; //7
         4'd8: Cnode<= 7'b0000000; //8
         4'd9: Cnode<= 7'b0000100; //9
         4'd10:Cnode<= 7'b0001000; //A
         4'd11:Cnode<= 7'b1100000; //B
         4'd12:Cnode<= 7'b0110001; //C
         4'd13:Cnode<= 7'b1000010; //D
         4'd14:Cnode<= 7'b1011000; //POSITIVE AKA 1110
         4'd15:Cnode<= 7'b0010100; //NEGATIVE AKA 1111
         default: Cnode = 7'b0000000; //DEFAULT CASE EVERYTHING ON
```

```
            endcase
        end
endmodule

`timescale 1ns / 1ps

module SEGDRIVE_TB(

    );
    reg [31:0] tmp_SW_tb;
    wire [6:0] Cnode_tb;
    wire dp_tb;
    wire [7:0] AN_tb;


    SEGDRIVE COMP (
    .tmp_SW(tmp_SW_tb),
    .Cnode(Cnode_tb),
    .dp(dp_tb),
    .AN(AN_tb)

    );

    initial
        begin
        //dp_tb = 1;
        tmp_SW_tb=0;
        #10
        tmp_SW_tb=4;
        #1000
        $finish;
        end




endmodule
```

This code takes an input of 32 switches and depending on when the switches change value, one of the many possible outputs will be selected to display on the 7 segment display. AN and dp are assigned in the beginning to make sure it is only one digit without the decimal place being shown. The different cases range from 0 to 9, and the 32-bit switch value selects the specific digit that needs to be modified. In this particular code 0 represents on, while 1 represents off. The values of E and F are set to display the positive and negative “sign” that is desired from the alu.

```
        `timescale 1ns / 1ps
```

```
module top(
    //INPUTS FROM FPGA
    input [3:0] opa,
    input [3:0] opb,
    input signa,
    input signb,
    input [1:0] asm,
    //OUTPUT TO 7SEG DISPLAY
   // output [7:0] opc,
    //output signc1,
    //7SEG DISPLAY INPUTS
    input clkt,
    output [6:0] Cnode1,
    output [7:0] AN1,
    output seg
    );

    wire [31:0] seg7Digit;
    //Send input changes to 7seg display
    assign seg7Digit[3:0] = opb;
    assign seg7Digit[7:4] = signb + 14;
    assign seg7Digit[11:8] = opa;
    assign seg7Digit[15:12] = signa + 14;
    assign seg7Digit[27:24] = signc1 + 14;
    assign seg7Digit[31:28] = asm;

    wire [7:0] opc;

    ALU alu_wrapper(
     .opa(opa),
     .opb(opb),
     .signa(signa),
     .signb(signb),
     .asm(asm),
     .opc(opc),
     .signc1(signc1)
    );


    bin2bcd b2bwrap(
    .bin(opc),
    .bcd(seg7Digit[23:16])
   );

    SEGDRIVE seg7driver_wrapper(
        .nexysCLK(clkt),
        .inDigit(seg7Digit),
         .Cnode(Cnode1),
        .dp(seg),
        .AN(AN1)
    );
```

```
endmodule

`timescale 1ns / 1ps
module top_tb();
    reg [3:0] opatb;
    reg [3:0] opbtb;
    reg signatb;
    reg signbtb;
    reg [1:0] asmtb;

    wire [6:0] Cnode_tb;
    wire dp_tb;
    wire [7:0] AN_tb;


    top top_t(
        .opa(opatb),
        .opb(opbtb),
        .signa(signatb),
        .signb(signbtb),
        .asm(asmtb),
        //output to 7seg
         .Cnode1(Cnode_tb),
         .seg(dp_tb),
         .AN1(AN_tb)

    );
    initial
        begin: TEST
            asmtb = 3;
            opatb = 4;
            opbtb = 2;
            signatb = 0;
            signbtb = 0;

            #100

            asmtb = 2;
            opatb = 4;
            opbtb = 4;
            signatb = 0;
            signbtb = 1;

            #100

            asmtb = 1;
            opatb = 9;
            opbtb = 1;
            signatb = 0;
            signbtb = 0;

            #100
            asmtb = 3;
```

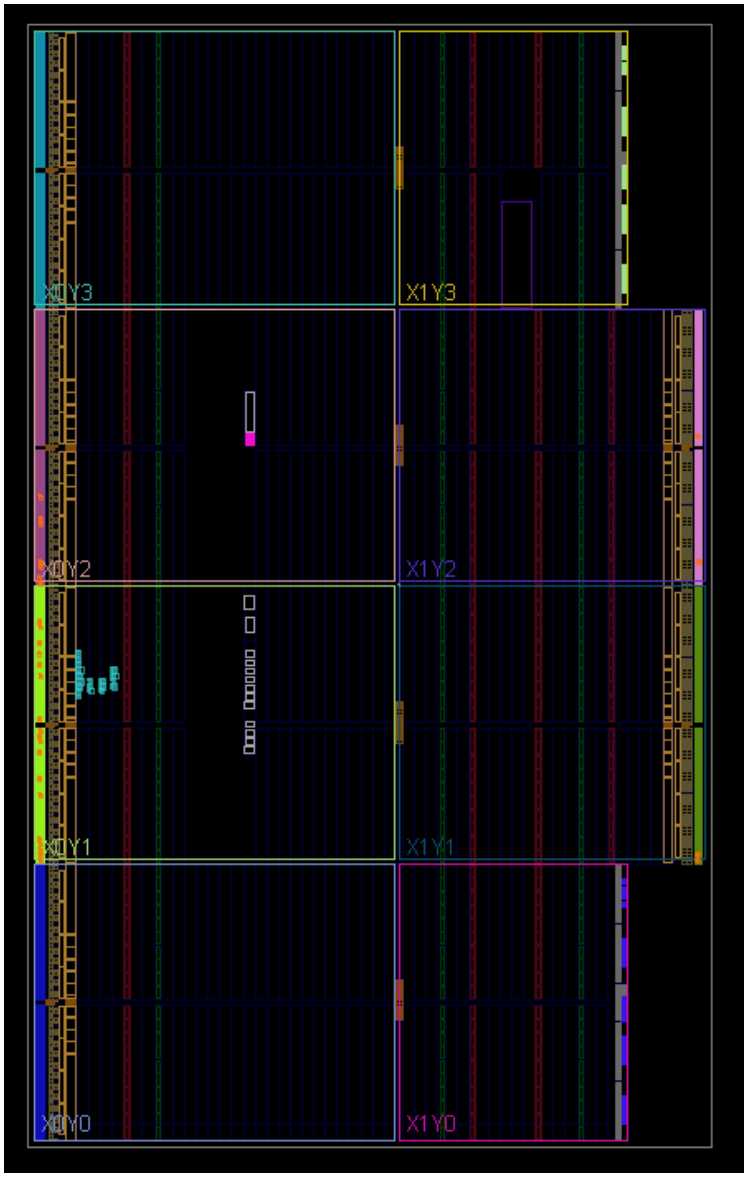
```
        opatb = 1;
        opbtb = 9;
        signatb = 1;
        signbtb = 0;
        #1000
        $finish;
    end
endmodule
```

The top file instantiates the ALU input, 8-bit output of the BCD converter from the ALU output, and the SEGDRIVE input, Cnode, dp, and Anode. The wires are used to establish connections between the ALU output to the binary to bcd converter, and also establish the input and output connections to the SEGDRIVE. The assign is used to give specific digits on the FPGA the desired values, whether it's an actual number or sign.

Corner Cases/Error :

One corner case can be A-D can be displayed, to cope this either the default case in the case statements should be 9, or the user simply just shouldn't input those values.

IMPLEMENTED DESIGN/ TIMING SUMMARY:



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.638 ns	Worst Hold Slack (WHS):	0.336 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	64	Total Number of Endpoints:	64	Total Number of Endpoints:	34

All user specified timing constraints are met.

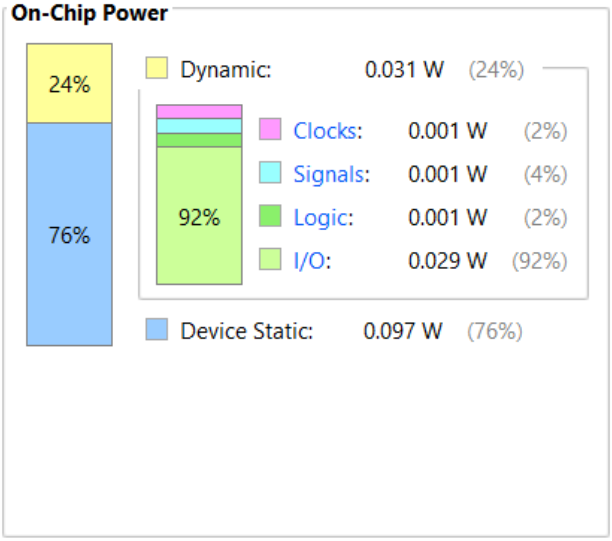
POWER SUMMARY:

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.129 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (12.9 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



UTILIZATION:

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)	Bonded IOB (210)	BUFGCTRL (32)
N top	75	36	5	31	75	29	1
alu_wrapper (ALU)	25	0	5	10	25	0	0
seg7driver_wrapper (SEGDRIVE)	28	36	0	18	28	0	0

RESOURCE USAGE:

WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF
								76	36
4.638	0.000	0.336	0.000		0.000	0.129	0	75	36